UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Corso di Laurea in Informatica, Big Data

Relazione attività Big Data

Studente:

Lorenzo Stigliano
Matricola 136174

Anno Accademico 2022-2023

Contents

1 Contenuto della relazione

La relazione si pone l'obiettivo di fornire al lettore una panoramica su una tecnologia NoSQL ad alte performance che negli ultimi anni si è fatta spazio all'interno del panorama dei DBMS e database, nonché soluzioni per il cloud, trovando impiego anche in grandi aziende come: Disney, Discord, Paloalto, Comcast e molte altre. Si sta parlando di ScyllaDB. La struttura del documento prevede un capitolo introduttivo2 su ScyllaDB, che racconta come è nata l'idea del suddetto progetto e di come funzioni a grandi linee. A seguire una analisi tecnica sul funzionamento di ScyllaDB 3, che è diviso in diverse sottosezioni:

1. Topologia 3.1: in cui viene analizzata la struttura di ScyllaDB.
2. IO Scheduler 3.2: approfondimento sul sistema di scheduling per le operazioni input e output.
3. Struttura dei Dati 3.3: analisi sul modo in cui i dati vengono salvati all'interno del database.
4. Metodi di Interrogazione 3.4: linguaggio di interrogazione per ScyllaDB.
5. Implementazione in Python 3.5: utilizzo e funzionamento della libreria python preposta all'interazione con cluster basati su ScyllaDB.

In alcuni di questi capitoli viene effettuato un confronto diretto con alcune delle peculiarità di alcune delle tecnologie più famose viste durante il corso.

2 Introduzione

Il progetto nasce da due ex programmatori di Red-Hat: Avi Kivity e Dor Laor durante lo sviluppo di un sistema operativo open source orientato al cloud computing, chiamato OSv (rilasciato il 16/09/2013 sotto licenza AGPL), all'interno di Cloudius Systems [9].

Figure 1: Avi Kivity e Dor Laor

L'idea è quella di unire il mondo delle programmazione ad alte performance alla potenza del cloud, sviluppando un sistema operativo *ad hoc* che fornisca tutti gli strumenti per la gestione di macchine virtuali e server, avendo il minimo overhead possibile.

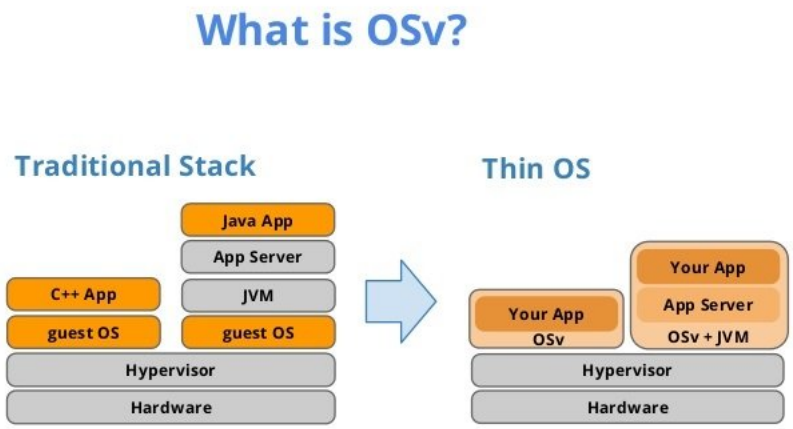

Figure 2: Stack OSv

All'inizio del progetto OSv veniva utilizzato come DBMS Cassandra. Si tratta di un progetto open source Top-Level (come anche CouchDB e Apache

HTTP Server), sviluppato da Apache Software Foundation per gestire grandi quantità di dati dislocati in diversi server, fornendo inoltre un servizio orientato alla disponibilità, senza alcun point of failure.

Figure 3: Logo CassandraDB

È una soluzione NoSQL che inizialmente fu sviluppata da Facebook. Jeff Hammerbacher, che ha guidato il team di Facebook, ha descritto Cassandra come un modello di dati simile a BigTable in esecuzione su una infrastruttura tipi Amazon-Dynamo. Cassandra fornisce una struttura di memorizzazione chiave-valore, con Eventual Consistency.

Alle chiavi corrispondono dei valori, raggruppati in famiglie di colonne: una famiglia di colonne è definita quando il database viene creato. Tuttavia le colonne possono essere aggiunte a una famiglia in qualsiasi momento.

Inoltre, le colonne sono aggiunte solo specificando le chiavi, così differenti chiavi possono avere differenti numeri di colonne in una data famiglia. I valori di una famiglia di colonne sono memorizzati insieme, in quanto Cassandra adotta un approccio ibrido tra DBMS orientato alle colonne e la memorizzazione orientata alle righe. Sicuramente l'aspetto che più di tutti ha penalizzato l'utilizzo Cassandra è il fatto che non avesse delle performance che fossero in linea con gli intenti di OSv. Da qui l'idea di creare un nuovo progetto il cui punto di forza fossero le performance, ma che mantenesse gli aspetti positivi di Cassandra:

1. decentralizzazione: i nodi nel cluster sono identici. Non esiste alcun single point of failure
2. full-tollerance: i dati vengono replicati automaticamente su più nodi. È supportata la replica mediante diversi data center, e la sostituzione dei nodi può essere effettuata senza alcun downtime
3. tunable consistency: i nodi nel cluster sono identici. Non esiste alcun single point of failure
4. Elasticità: il throughput di lettura o scrittura scala linearmente con l'aggiunta di nuove macchine (nodi), senza downtime e senza interruzione di alcun applicativo

Nasce così nel 2014: ScyllaDB, un database open-source di tipo distribuito wide-column store. Come suggerisce la tipologia di database non è di tipo

relazionale, ma utilizza un sistema chiave-valore per la gestione dei dati. Implementa gli stessi protocolli di cassandra, ma essendo basato su C++20, anziché su Java, ha delle performance nettamente superiori. Il fatto che sia scritto in un linguaggio di livello più basso rispetto a Cassandra porta con sé che il codice esegua più velocemente e si possa gestire una ottimizzazione con l'hardware migliore rispetto a Java. Rispetto al suo antesignano sono state inoltre sostituite librerie storiche, come Seastar, a favore di un utilizzo più aderente alle librerie che oggi sono disponibili su dispositivi basati su kernel linux.

Figure 4: Logo ScyllaDB

3 Analisi Tecnica

3.1 Topologia

La topologia di ScyllaDB è costituita in maniera circolare, per cui ogni nodo è perfettamente identico agli altri in termini di funzionalità (hanno tutti capacità di lettura e scrittura). Non siamo più in presenza di un paradigma piramidale, come nel caso di MongoDB, in cui solamente il nodo *primary* è in grado di eseguire operazioni di lettura e scrittura.
Ogni nodo gestisce una porzione di dati e se necessità di informazioni che non possiede, esegue una richiesta esplicita di tali informazioni verso altri nodi. Questo tipo di topologia semplifica, come vedremo, la gestione del carico di lavoro tra i nodi, poiché non abbiamo bisogno di un sistema che effettui un continuo bilanciamento.

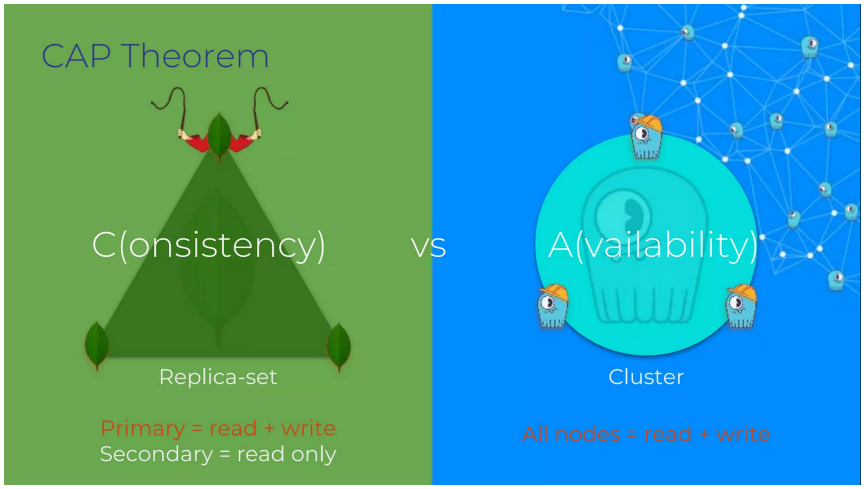

Figure 5: Topologia ScyllaDB in confronto con MongoDB

Creare dei cluster attraverso il paradigma proposto da ScyllaDB è più semplice rispetto ad altre tecnologie, come per esempio in MongoDB, in cui dobbiamo aggiungere un nodo primario per creare un nuovo cluster, ma questo porta potenzialmente ad una divisione dei dati e dovremmo quindi utilizzare un strumento che conosca la posizione esatta di tutti i dati a disposizione suddivisi nei vari elementi del cluster.

Oltre a questo aspetto è necessario utilizzare un sistema di balancing del carico di lavoro attraverso tutti gli elementi del cluster così organizzati. Si ottiene così un considerevole overhead in termini di implementazione, rispetto a ciò che avviene se si effettuassero operazione di clustering con ScyllaDB.

In ScyllaDB posso aggiungere nodi senza dovermi preoccupare della loro implementazione e del cambio della topologia del sistema. Immaginate quanto, in particolare su larga scala, possa essere conveniente utilizzare un sistema con una gestione dei cluster fatta con un sistema con una topologia come quella proposta da ScyllaDB.

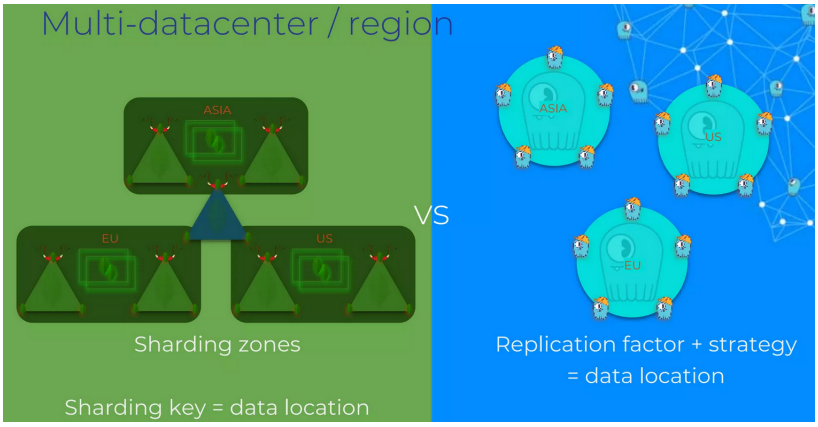

Figure 6: Gestione cluster di ScyllaDB con MongoDB

Come accennato nel paragrafo precedente: MongoDB, in situazione di multipli datacenter, utilizza un backgorund job che effettua balancing del carico di lavoro generando latenze in termini di perfomance. Un modo in cui gli sviluppatori di Mongodb hanno pensato di mettere di imporre al balancer di non operarare all'interno di alcune finestre temporali ben precise, come le ore in cui si è a pieno regime di lavoro, per evitare ulteriori latenze e disservizi.

Un aspetto importante di ScyllaDB è che applichi la propria topologia all'hardware che viene utilizzato, mappando la sua struttura su tutti i core disponibili. Questo permette di non avere bisogno di un balancer del workload sui nodi, ma, come verrà approfondito nel capitolo successivo, è applicato uno scheduler che opera specificatamente per le operazioni di lettura e scrittura.

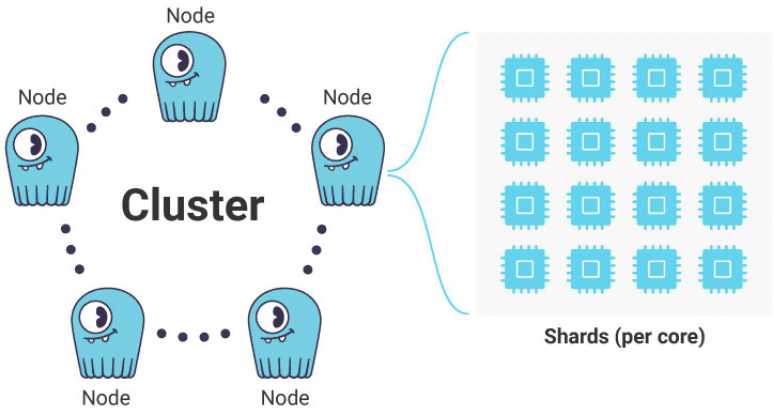

Figure 7: Un nodo per singolo core

3.2 IO Scheduler

Come anticipato al termine del capitolo precedente, un aspetto molto importante per ScyllaDB è lo scheduler I/O. All'interno del progetto non viene utilizzato lo scheduler del kernel Linux per gestire operazioni di lettura e scrittura, ma viene utilizzato Seastar. Questa è una libreria scritta in C++ che permette di scrivere applicazioni complesse ed estremamente efficienti per server multi-core. Generalmente le librerie e i frameworks per scrivere applicazioni per server si dividono in due macro categorie:

1. orientate all'efficienza
2. orientate alla complessità

Ci sono librerie che permettono di creare applicazioni che siano estremamente efficienti, ma molto semplici, o viceversa esistono librerie per costruire applicazioni complesse, ma in maniera poco efficiente. Seastar si pone a metà tra queste due categorie, offrendo un framework di programmazione completamente asincrono; si pone inoltre l'obiettivo di gestire ogni tipo di evento asincrono: network I/O, disk I/O e la combinazioni di questi ed altri eventi che possono occorrere in una applicazione per server.

Da quando sono nate le architetture multi-core oppure multi-socket, si è sempre sofferta la penalità di condividere i dati tra i vari cores in termini di latenza e di consistenza dei dati. Il progetto approccia questo problema imponendo una politica denominata "shared-nothing programming", in cui la memoria disponibile è divisa tra i core e ognuno lavora sui dati a disposizione nella regione di memoria assegnata. Qualora un core avesse bisogno di un dato che non possiede viene inviata una richiesta esplicita agli altri cores per ottenere l'informazione necessaria.

Tutto questo viene effettuato all'interno dello userspace. Questa scelta è stata fatta dagli sviluppatore per ottenere ancora maggiore controllo. Il sistema effettua lo scheduling I/O per ottenere il maggior throughput possibile, ma il kernel potrebbe effettuare un riordinamento sulla coda dei task appena creata. Creando la coda all'interno dello userspace si evita questa situazione e si fornisce al kernel solamente il minimo delle informazioni possibili al fine di ottenere il miglior risultato possibile.

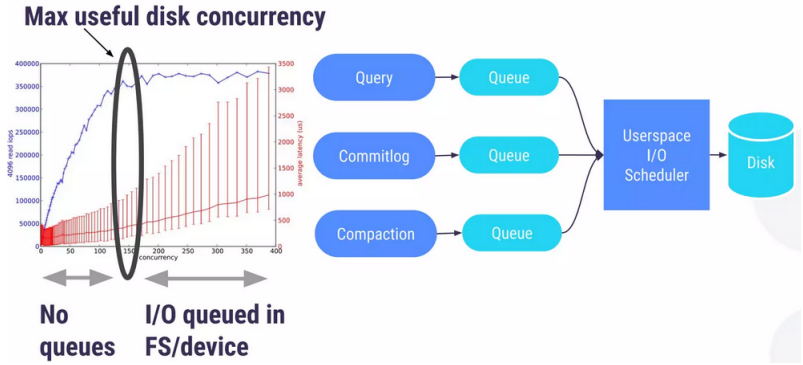

Figure 8: Schema funzionamento scheduler Seastar

3.3 Struttura dei Dati

ScyllaDB è un database NoSQL di tipo column-oriented. In questo tipo di databse i dati vengono rappresentati sfruttando le colonne con un layout di questo tipo:

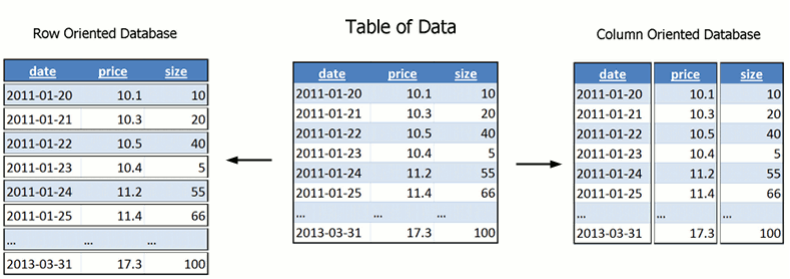

Figure 9: Schama funzionamento scheduler Seastar

Questo ci fornisce vantaggi in termini di tempi di accesso e compressione dei dati su disco (qualora si intendesse effettuare una compressione dei dati). In un database di tipo colonnare vengono effettuate meno letture del disco per poter ottenere le stesse informazioni che otterremmo in più letture in altri tipi di database. Questo è estremamente visibile nel momento in cui abbiamo grandi quantità di dati, considerando che in questi contesti il collo di bottiglia maggiore sono spesso le operazioni di lettura e scrittura su disco (in particolare se ci troviamo a dover utilizzare un disco meccanico).

Date le peculiarità dei database colonnari sono perfettamente applicabili in contesti OLAP. OLAP (On-Line Analytical Processing) indica un insieme di tecniche software per l'analisi interattiva e veloce di grandi quantità di dati, che è possibile sottoporre ad analisi di qualsiasi tipo. Questa è la componente tecnologica base del data warehouse. Alcuni esempi di analisi concreti possono essere:

1. l'analisi dei risultati delle vendite;
2. l'andamento dei costi di acquisto delle merci;
3. valutazioni di efficacia rispetto alle metodologie di marketing;
4. organizzazione dei dati per sondaggi;

In questi contesti sono estremamente importanti aspetti come: performance, scalabilità, capacità di compressione dei dati, etc. ... e un database di questo tipo risponde positivamente a questi requisiti. Il limite maggiore di questo tipo di tecnologia sono i tempi di scrittura, quindi l'inserimento di nuovi dati, in particolare rispetto ad altri sistemi come quelli che si basano sulle righe. Pur avendo un caso d'uso che non è specifico è possibile ibridare questa soluzione

con altre ed ottenere il miglior risultato possibile in termini di performance, soprattutto in situazioni in cui dobbiamo fare operazioni sui dati miste.

3.4 Metodi di Interrogazione

Per andare ad effettuare delle query viene utilizato lo stesso sistema per Cassandra (CQL = Cassandra Query Language), integrato con delle estensione per poter funzionare specificatamente con ScyllaDB. CQL è simile a SQL (Structured Query Language), usato per operare con database relazionali, i quali descrivono i dati in tabelle organizzate per colonne e righe. In un sistema come quello di MongoDB possiamo fare query in maniera molto più flessibile ed efficiente:

1. geospatial queries
2. text search
3. aggregation pipelines
4. graph queries
5. change streams

Mentre in un sistema columnar-based, come ScyllaDB, abbiamo molte meno possibilità in termini di interrogazioni possibili, ma in compenso si mantengono delle performance uniformi, che per contesti specifici, come quello del real time cloud computing, è molto importante.

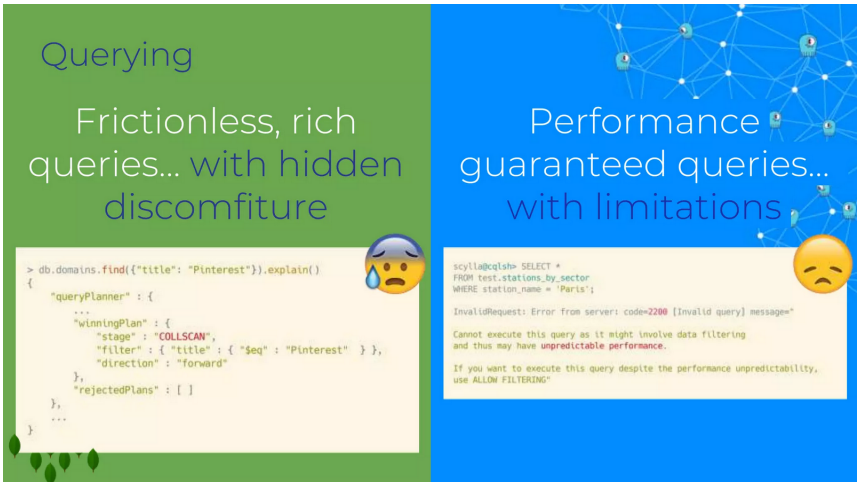

Figure 10: Confronto tra interrogazioni su ScyllaDB e MongoDB

ScyllaDB può anche essere usato insieme ad altre tecnologie SQL oppure NoSQL e diventare parte di un ecosistema più vasto, grazie all'integrazoine con altre piattaforme opensource quali: Apache Kafka (piattaforma open source di stream processing scritta in Java e Scala) e Apache Spark (framework open source per il calcolo distribuito).

3.5 Implementazione in Python

Esiste la possibilità di gestire il proprio database attraverso il linguaggio di programmazione python; questo avviene grazie a delle librerie preposte a questo compito come: pymongo (per quel che riguarda MongoDB) oppure python-driver (per ScyllaDB). Sicuramente l'implementazione di pymongo è ad uno stato più maturo rispetto a quella di pyton-driver ed inoltre gli sviluppatori di pymongo hanno sviluppato delle altre librerie per altri linguaggi che usano in maniera molto uniforme metodi con nomi e funzionamento identici tra di loro.

Per installare la libreria e poter effettuare la connessione ad un cluster ScyllaDB è sufficiente utilizzare il seguente comando:`pip install scylla-driver` (presupponendo di avere python $\geq$ 2.7 installato). Per cominciare ad utilizzare il driver è necessario effettuare il setup iniziale ed eseguire la connessione nei confronti di un cluster. All'interno del codice troveremo varie volte citato il nome Cassandra, questo perché, come visto nei capitoli precedenti, è strettissimo il legame che intercorre tra i due anche in termini di driver utilizzati. Di fatto il `scylla-driver` è perfettamente compatibile con tutti gli strumenti che Cassandra ha in dote e molti di questi vengono ampiamente utilizzati per il corretto funzionamento dello stesso driver di ScyllaDB. Il modo più semplice per connettersi ad un cluster è il seguente:

```
1 from cassandra.cluster import Cluster
2
3 cluster = Cluster()
4
```

In questo modo stiamo creando una connessione in locale (127.0.0.1), ma è anche possibile specificare un indirizzi IP o più indirizzi, se vogliamo connetterci ai più nodi con la seguente opzione: `cluster = Cluster(['192.168.0.1', '192.168.0.2'])`. Se abbiamo bisogno di utilizzare una porta specifica possiamo configurare l'istruzione di connessione come segue: `cluster = Cluster(['192.168.0.1', '192.168.0.2'], port=..., ssl_context=...)`. L'indirizzo IP che stiamo fornendo al driver funge unicamente da punto di accesso; questo in automatico andrà ad esplorare il cluster connettendosi a tutti i nodi presenti non appena andremo a creare una sessione:

```
1
2 session = cluster.connect()
3
```

Una volta fatto ciò potremmo iniziare a lavorare sul cluster. E' possibile fornire alla sessione un *keyspace* passandolo come parametro: `session = cluster.connect('mykeyspace')`. L'utente può cambiare in qualsiasi momento questo parametro attraverso altri due metodi: `session.set_keyspace('user')` oppure `session.execute('USE users')`.

Esiste la possibilità di andare a modificare il profilo della sessione in corso specificando le regole di balancing, consistenza dei dati, timeout delle richieste, etc. ... con la seguente sintassi:

```

from cassandra.cluster import Cluster, ExecutionProfile,
EXEC_PROFILE_DEFAULT
from cassandra.policies import WhiteListRoundRobinPolicy,
DowngradingConsistencyRetryPolicy
from cassandra.query import tuple_factory

profile = ExecutionProfile(
    load_balancing_policy=WhiteListRoundRobinPolicy(['127.0.0.1']),
    retry_policy=DowngradingConsistencyRetryPolicy(),
    consistency_level=ConsistencyLevel.LOCAL_QUORUM,
    serial_consistency_level=ConsistencyLevel.LOCAL_SERIAL,
    request_timeout=15,
    row_factory=tuple_factory
)
cluster = Cluster(execution_profiles={EXEC_PROFILE_DEFAULT: profile})
session = cluster.connect()

print(session.execute("SELECT release_version FROM system.local").one())

```

Per effettuare delle interrogazioni si consiglia di utilizzare la seguente sintassi utilizzando il comando `execute`:

```

query = session.execute('SELECT name, age, email FROM users')
for user in query:
    print user.name, user.age, user.email

```

A meno di configurazioni differenti da parte dell'utente i risultati delle ricerche sono ritornati sotto forma di *namedtuple*. Queste si possono navigare in diversi modi, se ne elencano di seguito ulteriori due equivalenti al precedente esempio:

```

query = session.execute('SELECT name, age, email FROM users')
for (name, age, email) in query:
    print name, age, email

query = session.execute('SELECT name, age, email FROM users')
for data in query:
    print data[0], data[1], data[2]

```

Esiste la possibilità di creare delle interrogazioni che vengano eseguite da Cassandra e poi salvate per un secondo momento. Questo riduce il traffico di rete e l'impatto sulla CPU. Per effettuare questo tipo di interrogazioni viene utilizzato il metodo `prepare()` con la seguente sintassi:

```
1
2  user_lookup_stmt = session.prepare("SELECT * FROM users WHERE user_id=?")
3
4  users = []
5  for user_id in user_ids_to_query:
6      user = session.execute(user_lookup_stmt, [user_id])
7      users.append(user)
8
```

Si possono anche effettuare operazioni in linguaggio *COQL* citato nel capitolo sui metodi di interrogazione 3.4 utilizzando il metodo `execute`:

```
1  // %s è solamente un placeholder
2  session.execute(
3      """
4      INSERT INTO users (name, credits, user_id)
5      VALUES (%s, %s, %s)
6      """,
7      ("John O'Reilly", 42, uuid.uuid1())
8  )
9
```

Se l'utente desidera eseguire delle interrogazioni in maniera asincrona è necessario utilizzare il metodo `execute_async()` che ritorna un oggetto denominato *ResponseFuture*. Partendo da questo oggetto esistono due modi per poter ottenere il valore dell'interrogazione: utilizzando il metodo `result()` come segue:

```
1  # build a list of futures
2  futures = []
3  query = "SELECT * FROM users WHERE user_id=%s"
4  for user_id in ids_to_fetch:
5      futures.append(session.execute_async(query, [user_id])
6
7  # wait for them to complete and use the results
8  for future in futures:
9      rows = future.result()
10     print rows[0].name
11
```

Oppure agganciare all'interrogazione uno dei seguenti metodi `add_callback()`, `add_errback()`, `add_callbacks()`. Di seguito un esempio del metodo `add_callbacks()`:

```
1  def handle_success(rows):
2      user = rows[0]
```

```
    try:
        process_user(user.name, user.age, user.id)
    except Exception:
        log.error("Failed to process user %s", user.id)
        # don't re-raise errors in the callback

def handle_error(exception):
    log.error("Failed to fetch user info: %s", exception)


future = session.execute_async(query)
future.add_callbacks(handle_success, handle_error)

```

Si sconsiglia di eseguire in maniera asincrona operazioni che tendono ad avere un tempo di esecuzione molto lungo perché potrebbe creare rallentamenti all'interno del sistema. Si rimanda alla documentazione ufficiale del driver per maggiori informazioni: [7]

Infine esiste la possibilità di modificare i *consistency level*; questo influisce il modo in cui una interrogazione viene ritenuta di successo. Il valore che è applicato solitamente all'opzione *ConsistencyLevel* è *LOCAL_ONE*, ma è possibile modificare tale valore come nel seguente esempio:

```
from cassandra import ConsistencyLevel
from cassandra.query import SimpleStatement

query = SimpleStatement(
    "INSERT INTO users (name, age) VALUES (%s, %s)",
    consistency_level=ConsistencyLevel.QUORUM)
session.execute(query, ('John', 42))

```

In questo caso si sta forzando l'interrogazione ad essere estesa a quanti più nodi possibili per ricevere una approvazione corale sulla sua correttezza. Per maggiori informazioni sui livelli di *consistency* si invita il lettore a visitare il seguente link: `https://docs.scylladb.com/stable/cql/consistency.html`

References

[1] Documentazione ScyllaDB, https://docs.scylladb.com/stable/

[2] Scylladb optimizes database architecture to maximize hardware performance, Nishant Suneja https://ieeexplore.ieee.org/abstract/document/8738153

[3] Documentazione Column oriented Database, https://en.wikipedia.org/wiki/Column-oriented_DBMS

[4] Scylla Summit 2019, https://www.youtube.com/watch?v=ch1vRQBiXtI&t=689s

[5] ScyllaDB Wide Column Store video, https://www.youtube.com/watch?v=bTEfRmdBq7I

[6] Writing application for ScyllaDB video, https://www.youtube.com/watch?v=uBID7EaVDBQ

[7] Documentazione driver python per ScyllaDB, https://python-driver.docs.scylladb.com/stable/

[8] Documentazoine Seastar, https://docs.seastar.io/master/tutorial.html#introduction

[9] OSv main website, https://www.cloudius-systems.com/

[10] Apache Spark main website, https://spark.apache.org/

[11] Apache Kafka main website, https://kafka.apache.org/